# Macros

Macros are inserts with the following additions:

a) Named parameters.  
When the macro is defined, named parameters are defined. When called, each of those  
parameters must contain a value.

#macro XXYYZ(alpha, beta, delta)  
#apply XXYYZ("alpha value", "beta value", "delta value")

Anywhere in the code that 'alpha' is found, it is replaced with 'alpha value'.

MK – In FSH, the equivalent would probably be :

\* RuleSet XXYYZ($alpha, $beta, $delta)

then at the point of insertion:

\* insert XXYYZ("alpha value", "beta value", "delta value")

Question: What type of substitutions are allowed, for example, would we support text substitution?

\* description = "This profile contains $alpha components"

b) Global parameters

There are parameters that are defined globally, such as

* the id of the current profile
* the url of the current profile.
* many others [Kurt: such as?]

[MK - I think this is reasonable.]

* parameters that can be set on command line, which can be referenced in any macro just like named parameters.

[MK - This would imply Sushi accepting user parameters on the command line and have a way to refer to those parameters. This opens some questions, e.g., what happens when a parameter is expected, but not supplied?]

c) Redirection  
The output of a macro can be redirected to a non-fsh file.

The output file name can contain global and named parameters which will get expanded when the macro is called/executed (not when it is first defined)

I use this to collate info such as node graph commands I process externally to generate svg images, and also to allow embedded macros to generate the --intro implementation guide files for each profile.

#macro XXYYZ() > 'redirection path'

[MK – RuleSets couldn’t act this way since they are expanded in place and do not have a concept of“outputs. It seems like separate scripting outside of FSH would be the approach here. I’m not sure if we should or could do this.]

d) Once flags  
The **once** flag allows a macro to be called only once, even if it is executed multiple times. There are two flavors of the **once** flag:

* restrict calls to once per profile
* restrict the calls to once period.

Named Parameters can be a problem here, as multiple calls can be made with difference named parameter values, and only the first call will create output with its unique parameter values.

// this is expanded once per profile.

#macro once XXYYZ()

#end

// this is expanded once, no matter how many times it is called.

#apply once XXYYZ()

### Example:

Suppose that:

* Several macros are defined that add distinct sliced Observation.component items. These macros are sometimes used in a profile together, and in some profiles separately
* Each macro calls a common macro to add component slicing; this should only be called once per profile.

One way to handle this is to force the user to call this macro separately, once per profile, but this requires the user to manually add this and allows opportunities to not call it. So, a macro is defined that implements the required slicing commands, and it is called with the once flag set, so it only generates FSH output the first time it is called.

[MK – I’m not sure what underlying execution model is being assumed here. The idea of “calling” a macro corresponds to an insert, so the multiple calls isn’t really defined.

It also appears the example scenario can be handled by RuleSets that contain other RuleSets. If there are separate rule sets for each component, then RuleSets for different combinations of components can be created. Each profile can then define the proper components using a single **insert** rule. Does that not work?]

# Fragments

A fragment is kinda a cross between a profile and a macro. The purpose of a fragment is to define a profile fragment that:

1. can be added to multiple profiles (like a macro)
2. is intended to signal to code generators that each fragment can be treated like a separate 'thing' (class or interface).
3. breaks the output profile into separate pieces that are functionally unique, implemented in multiple profiles, and documented as separate 'thing'.

The fragments generate structure definition and other meta data allowing external code generation tools to recognize when a fragment has been defined, and where it is used. Programming artifacts can be generated from this info.

In generated artifacts (including implementation guide):

* The fragment would have a parent class (like Observation).
* It should generate an item in the implementation guide showing exactly what it implemented.
* It should generate documentation in the documentation guide indicating what profiles implement it.
* A profile can include one or more fragments just like a macro.
* No named parameters are allowed in a fragment call.

### Example

* + - * Create a fragment 'f' that has three Observation.component items
* Include the fragment in multiple profiles 'a', 'b', and 'c'. I generate a C# project that implements a class to process these profiles.
* Generate a method 'm' that can accept an interface 'f' that mimics the fragment 'f'.
* Create classes that mimic profiles 'a', 'b', and 'c', each of which implements interface 'f'.
* Call method 'm' with an instance of 'a', 'b', or 'c'.

[MK – Could this be accomplished by creating an abstract profile with those three components that can be a direct parent of other profiles, or incorporate a common RuleSet? I am not understanding exactly why a code generator needs fragments to generate the correct interfaces.]

#Fragment: Observation.Component.NotPreviouslySeenFragment

#Parent: Observation

#Title: "NotPreviously Seen Fragment"

#Description: """

**Adds NotPreviously Seen Component slice.**

**"""**

#apply Observation.Component.Add("notPreviouslySeen", "0..\*", "Not Previously Seen", "Not Previously Seen",  
# """  
# This slice contains the optional components that define previous encounters in which this abnormality was not seen.  
# The value of this component is a codeable concept chosen from the NotPreviouslySeenVS valueset.  
# """,  
# "ObservationComponentSliceCodesCS#notPreviouslySeen")  
#  
#apply Observation.Component.SetConcept("notPreviouslySeen", "1..1", "NotPreviouslySeenVS")  
#end

# Conditional Compilation

This is completely experimental, and I have not found it to be useful so far, but there may be instances where it may become useful. This is identical to conditional compilation in C++, c3, etc.

We may want to ignore adding this to FSH. Not sure if it will ever be useful, though we still need some way to implement different countries using similar profiles.

#if (value1=="true")

if value1 is true, then include all text here.

#else if (value2=="false")

if value1 is false, then include all text here.

#else

otherwise include all text here.

#end

The original purpose of conditional compilation was to allow a profile to be made that could be compiled for different countries.

Profile: Hello

Parent: Observation

#if Country == "US"

• greeting = "Hello"

#else if Country == "CA"

• greeting = "Hello, eh"

#else if Country == "NO"

• greeting = "Uff da"

#else if Country == "AU"

• greeting = "G'Day"

#end